Modular SDN Programming w/ Pyretic

Joshua Reich

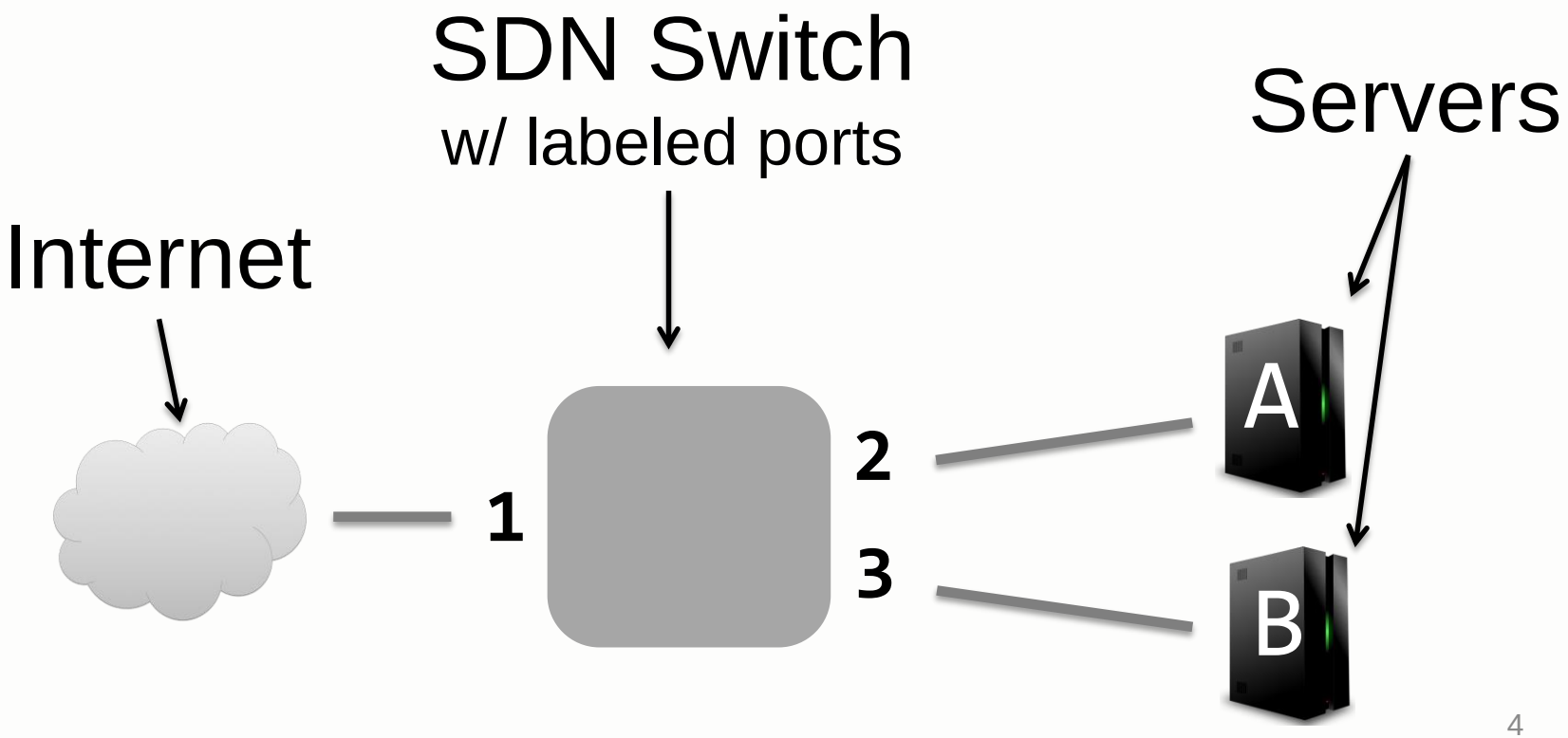
Princeton University

www.frenetic-lang.org/pyretic

SDN Is Great!

Programming w/ OpenFlow, Not So Much.

Example Network



A Simple OpenFlow Program

OpenFlow Priority Program

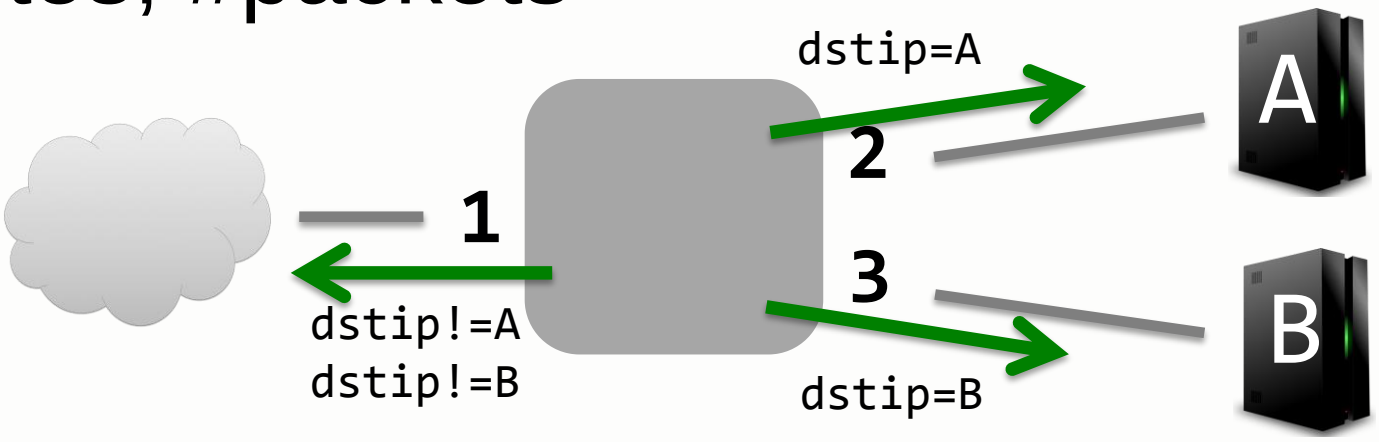
```
2:dstip=A   -> fwd(2)
1:*         -> fwd(1)
2:dstip=B   -> fwd(3)
```

Pattern

Action

Counters for each rule
- #bytes, #packets

Route: IP/fwd



One API, Many Uses

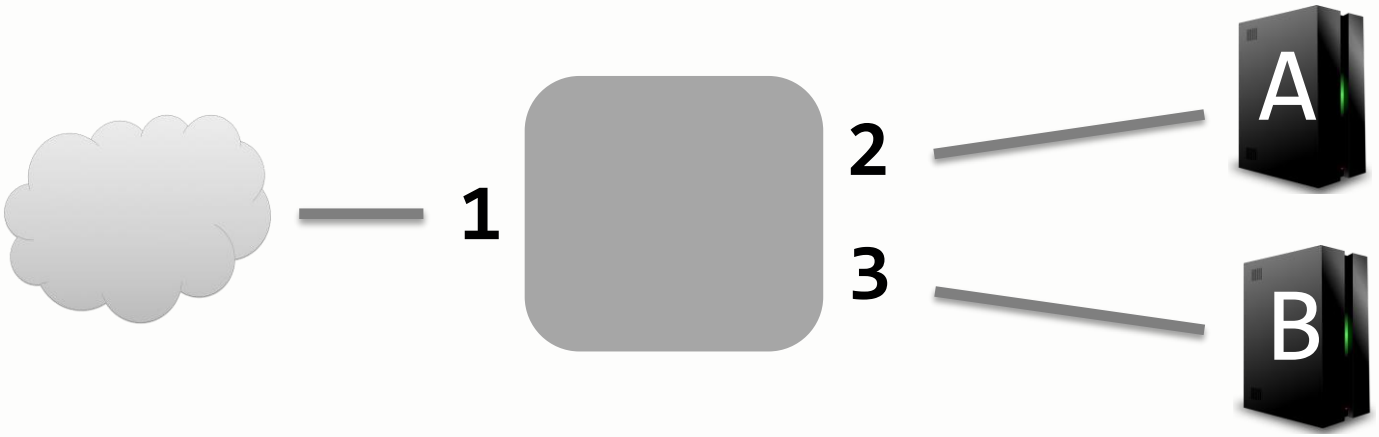
Priority Ordered

```
1:dstmac=A -> fwd(2)
1:dstmac=B -> fwd(3)
2:*        -> fwd(1)
```

Pattern

Action

***Switch*: MAC/fwd**



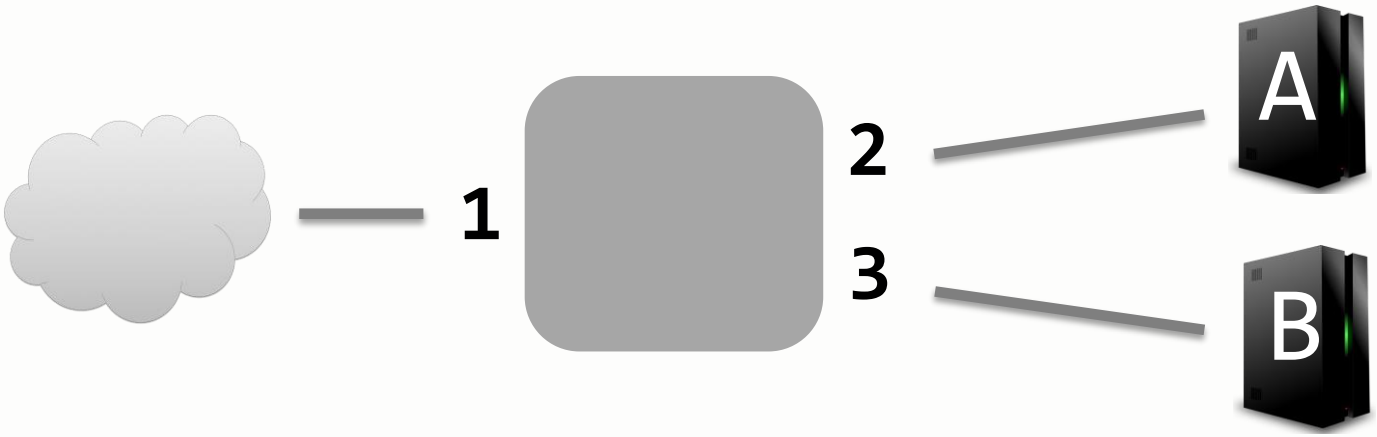
One API, Many Uses

```
srcip=0*,dstip=P -> mod(dstip=A)
srcip=1*,dstip=P -> mod(dstip=B)
```

Pattern

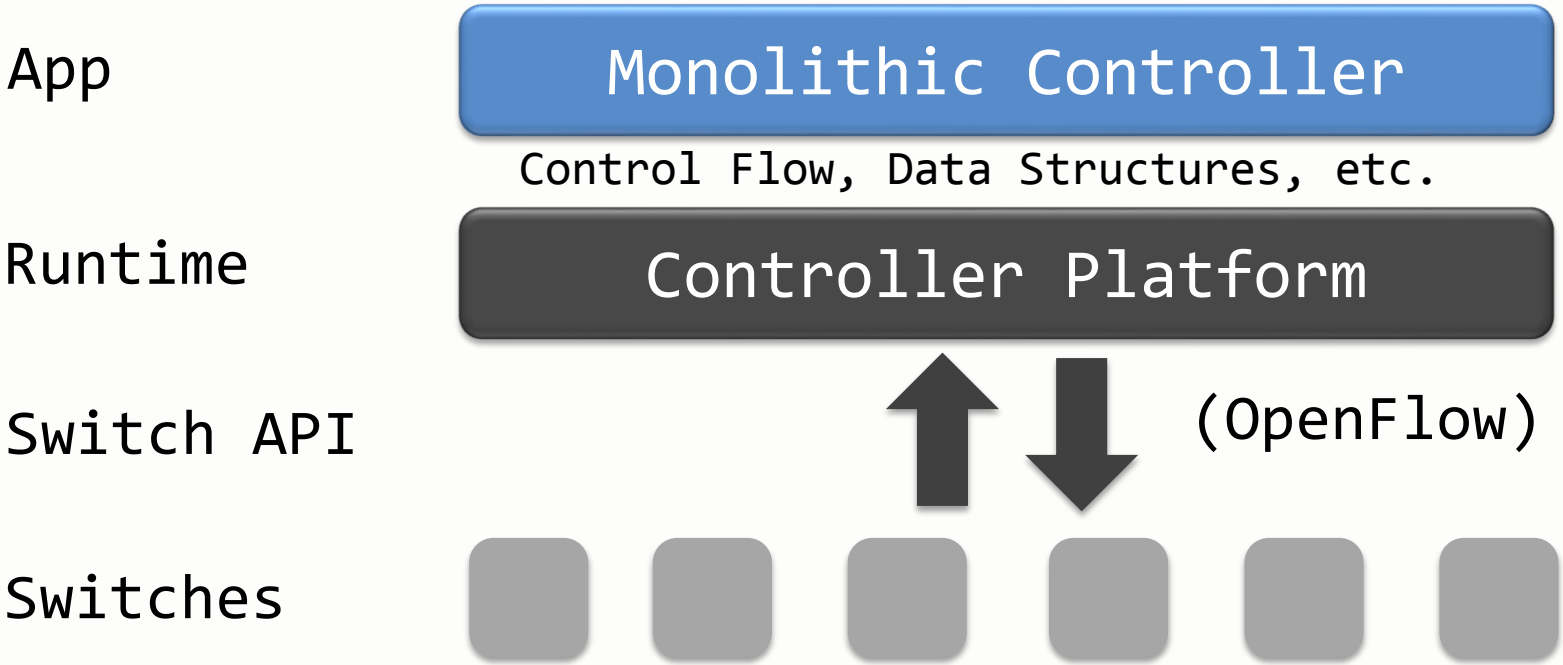
Action

***Load Balancer*: IP/mod**



OpenFlow Programming Stack

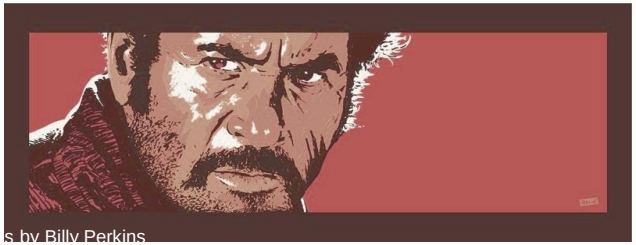
*First Order Approximation



Programming SDN w/ OpenFlow

- The Good
  - Network-wide visibility
  - Direct control over the switches
  - Simple data-plane abstraction

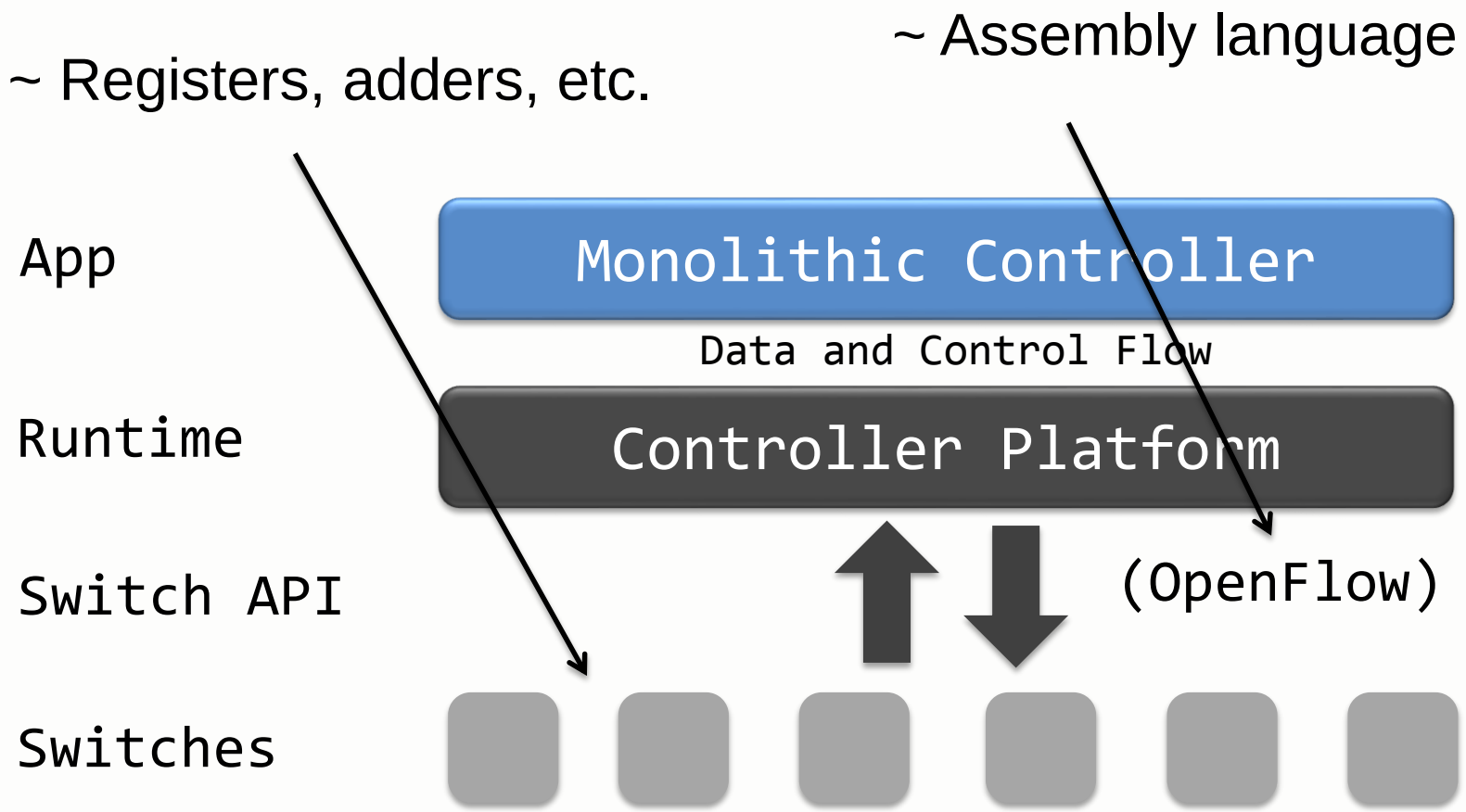
- The Bad
  - Low-level programming interface
  - Functionality tied to hardware
  - Explicit resource control



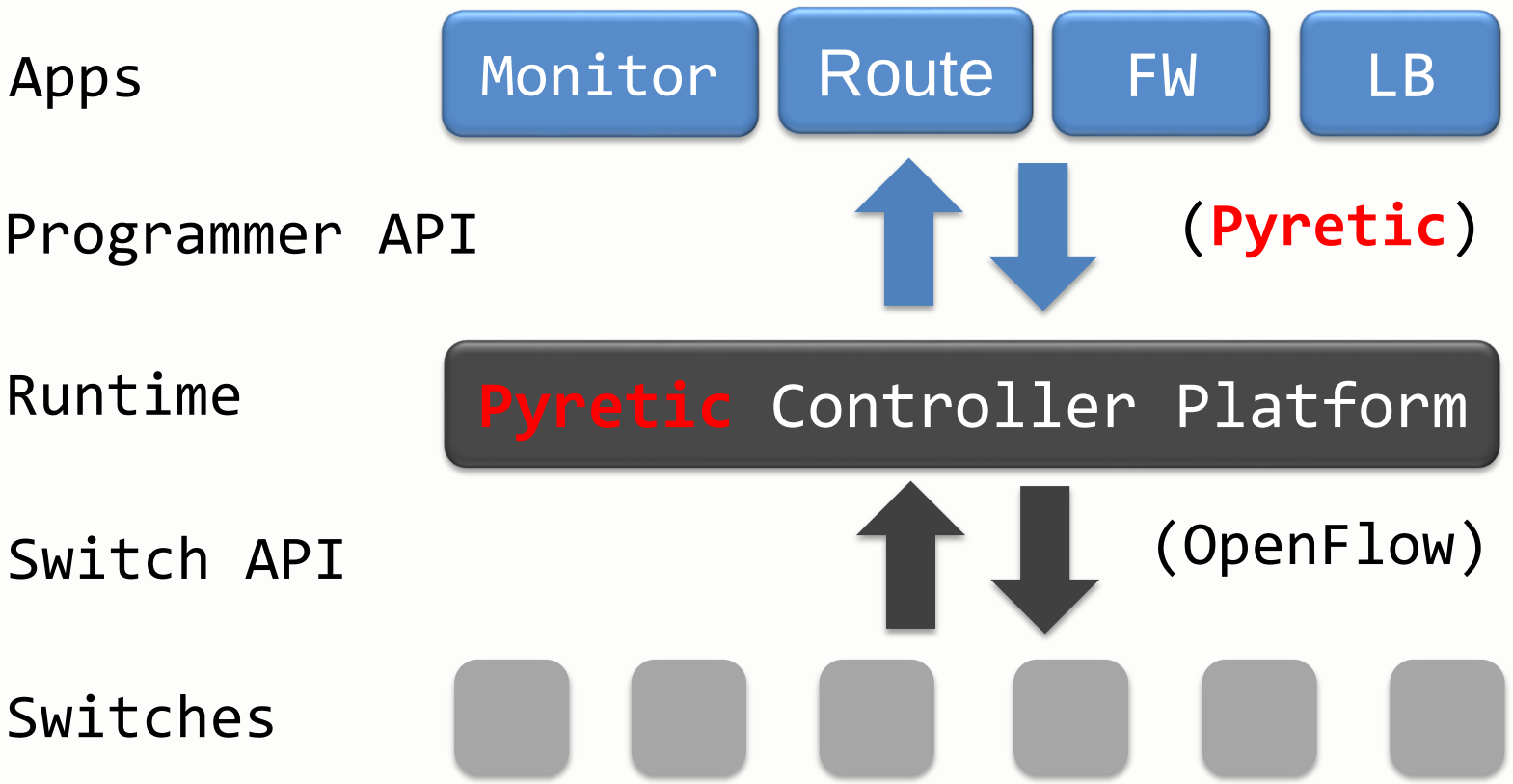
s by Billy Perkins

- The Ugly
  - Non-modular, non-compositional
  - Challenging distributed programming

Today: OpenDaylight, POX, etc.



Pyretic: Language & Platform



Pyretic from 10,000 ft

Pyretic Philosophy

- Provide high-level programming abstractions
- Based on state-of-the-art *PL* techniques
- Implement w/ state-of-the-art *systems* tech
- Package for the networking community
- Focus on real world utility and cutting edge features

Open Source and Programmer-Friendly

- Embedded and Implemented in Python
  - Rich support for dynamic policies
  - Familiar constructs and environment
  - Python library / module system
- Default 3-clause BSD license
- Active & growing developer community
  - GitHub repository & wiki
  - Video tutorials and documentation

Where Pyretic Fits

- Hard and soft-switches
  - Network processors
  - Reconfigurable pipelines
  - Hardware accelerated features
  - Programmable x86 dataplane
- **APIs and Languages**
- **Traffic Monitoring/Sampling/QoS**
- Testing, **Debugging**, Verification
- Experimentation & Testbeds
- **Integration & Orchestration**
  - End hosts
  - Legacy switches
  - Middleboxes
  - Network Services
- WAN, Enterprise, DC, Home
- Wireless & Optical
- **Systems Challenges**
  - Scalability
  - Fault-tolerance
  - Consistency
  - Upgradability
  - Performance
- Security
- **Virtualization**
  - of network
  - of services
  - of address space

Pyretic from 1,000 ft

Basic Programmer Wishlist

- Encode policy concisely
- Write portable code
- Specify traffic of interest
- Compose code from independent modules
- Query network state
- Modify policy dynamically

Pyretic =

* Our goal

Pyretic Provides

Feature	Example
Concise policy encoding	• `flood()`
Boolean predicates	• `~(match(dstip=10.0.0.1) | match(srcip=10.0.0.1) )`
Composition operators	• `(load_balance + monitor) >> route`
Virtual header fields	• `match(switch)` • `modify(class='staff')`
Query Policies	• `count_packets()`
Topology abstraction	• `merge` • `split` • `refine`

Pyretic from 500 ft

Policy in OpenFlow

- Defining “policy” is complicated
  - All rules in all switches
  - Packet-in handlers
  - Polling of counters
- Programming “policy” is error-prone
  - Duplication between rules and handlers
  - Frequent changes in policy (e.g., flowmods)
  - Policy changes affect packets in flight

From Rules to a Policy Function

- Located packet
  - A packet and its location (switch and port)
- Policy function
  - From located packet to set of located packets
- Examples
  - Original packet: `identity`
  - Drop the packet: `drop`
  - Modified header: `modify(f=v)`
  - New location: `fwd(a)`

Pyretic at sea-level

Why a set? Consider `flood`

To flood every packet:

```
from pyretic.lib.corelib import *

def main():
  return flood()
```

No worrying about:

- Writing a separate handler for controller
- Constructing and sending OpenFlow rules
- Keeping track of each switch
- Whether all switches supports the OpenFlow “flood” action (portability!)

Programmer Wishlist

- Encode policy concisely
- **Specify traffic of interest**
- Write portable code
- Compose code from independent modules
- Query network state
- Modify policy dynamically

From Bit Patterns to Predicates

- OpenFlow: bit patterns
  - No direct way to specify `dstip!=10.0.0.1`
  - Requires two prioritized bitmatches
    - Higher priority: `dstip=10.0.0.1`
    - Lower priority: `*`

- Pyretic: boolean predicates
  - Combined with `&`, `|`, and `~`
  - E.g., `~match(dstip=10.0.0.1)`

Example: Access Control

Block host 10.0.0.3

```
def access_control():
  return ~(match(srcip='10.0.0.3') |
           match(dstip='10.0.0.3') )
```

Programmer Wishlist

- **Encode policy concisely**
- **Write portable code**
- Specify traffic of interest
- Compose code from independent modules
- Query network state
- Modify policy dynamically

OpenFlow Packets

- Location specific code needs both
  - The packet
  - The OF packet_in message*
- Tagging packets requires choosing
  - Header field (e.g., VLAN, MPLS, TOS bits)
  - Set of values

Neither concise nor portable

* https://github.com/noxrepo/pox/tree/betta/pox/misc/of_tutorial.py

Pyretic Virtual Header Fields

- Unified abstraction
  - Real headers: `dstip`, `srcport`, …
  - Packet location: `switch` and `port`
  - User-defined: e.g., `traffic_class`
- Simple operations, concise syntax
  - Match: `match(f=v)`
  - Modify: `modify(f=v)`
- Example
  - `match(switch=A) & match(dstip='1.0.0.3')`

Runtime Handles Implementation

- Compiler chooses
  - What set of header bits to use
  - What each value means
- Conceptual – no tying to particular mech.
- Portable – different hw, other modules
- Efficient
  - Don’t need same vlan value network-wide
  - Sometimes tags will ‘factor out’

Programmer Wishlist

- Encode policy concisely
- Write portable code
- Specify traffic of interest
- **Compose code from independent modules**
- Query network state
- Modify policy dynamically

Recall OpenFlow

Balance then Route (*in Sequence*)

```
srcip=0*,dstip=P -> mod(dstip=A)
srcip=1*,dstip=P -> mod(dstip=B)
```

```
dstip=A  -> fwd(2)
dstip=B  -> fwd(3)
*        -> fwd(1)
```

Combined Rules?
(only one match)

```
srcip=0*,dstip=P -> mod(dstip=A)
srcip=1*,dstip=P -> mod(dstip=B)
dstip=A  -> fwd(2)
dstip=B  -> fwd(3)
*        -> fwd(1)
```

Balances w/o Forwarding!

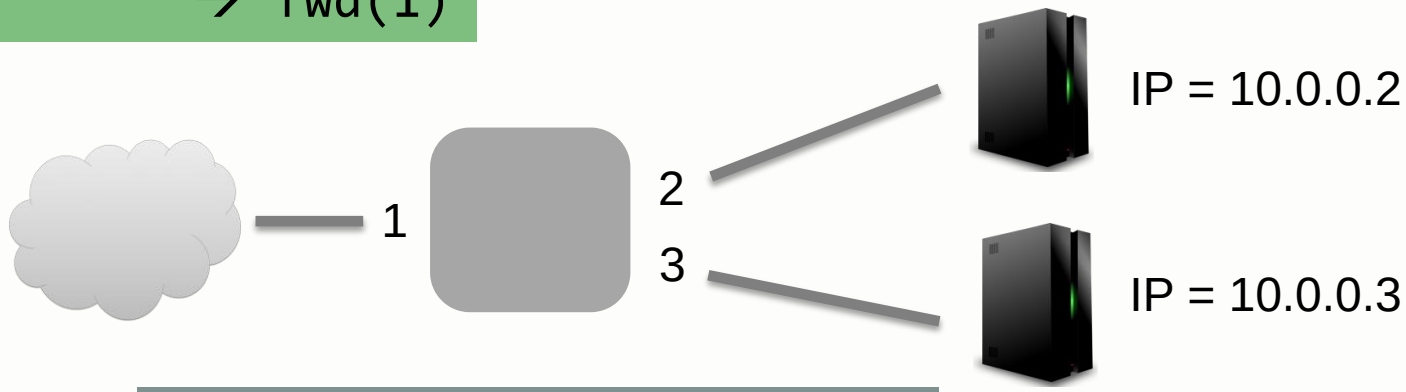
Route and **Monitor** (*in Parallel*)

Route

```
dstip = 10.0.0.2 → fwd(2)
dstip = 10.0.0.3 → fwd(3)
*                → fwd(1)
```

Monitor

```
srcip = 5.6.7.8 → count
```




```
srcip = 5.6.7.8  → count
dstip = 10.0.0.3 → fwd(3)
*                → fwd(1)
```

Forwards but doesn't count

Counts but doesn't forward

→ Combined rules installed on switch?

Requires a Cross Product

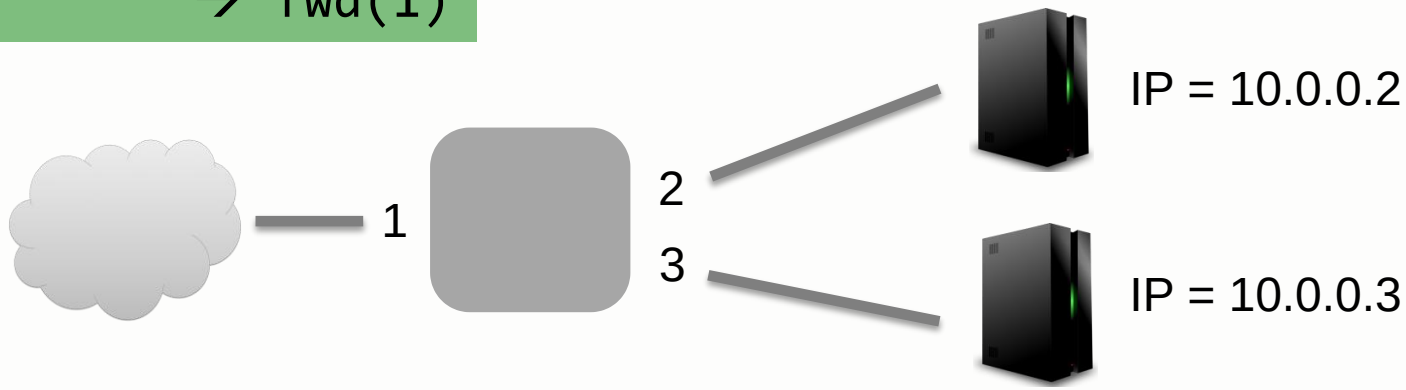
[ICFP'11, POPL'12]

Route

```
dstip = 10.0.0.2 → fwd(2)
dstip = 10.0.0.3 → fwd(3)
*                → fwd(1)
```

Monitor

```
srcip = 5.6.7.8 → count
```



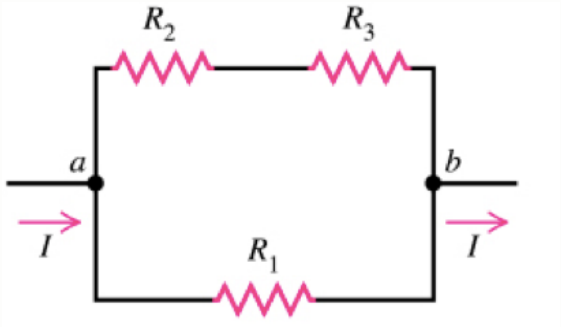

```
srcip = 5.6.7.8 , dstip = 10.0.0.2 → fwd(2) , count
srcip = 5.6.7.8 , dstip = 10.0.0.3 → fwd(3) , count
srcip = 5.6.7.8                    → fwd(1) , count
```

```
dstip = 10.0.0.2 → fwd(2)
dstip = 10.0.0.3 → fwd(3)
*                → fwd(1)
```

Policy Functions Enable Compositional Operators

Parallel '|': *Do both C1 and C2 simultaneously*

Sequential '>>': *First do C1 and then do C2*



Example: Sequential Composition

To first apply access control and then flood

```
access_control() >> flood()
```

- Two totally independent policies
- Combined w/o any change to either

Easily Define New Policies

E.g., the ‘if_’ policy

```
def if_(F,P1,P2):
  return (F >> P1) + (~F >> P2)
```

Or `flood`

```
xfwd(a) = ~match(inport=a) >> fwd(a)

flood =
  parallel([match(switch=sw) >>
            parallel(map(xfwd,ports))
            for sw,ports in MST])
```

- Portable: compiler chooses whether to implement using OpenFlow ‘flood’ or ‘forward’

Programmer Wishlist

- Encode policy concisely
- Write portable code
- Specify traffic of interest
- Compose code from independent modules
- **Query network state**
- Modify policy dynamically

Querying In OpenFlow

- Pre-install rules at the needed granularity (so byte/packet counters are available)
- Issue queries to poll these counters
- Parse the responses when they arrive
- Combine counter values from multiple rules
- Keep track of any packets sent to controller

Queries in Pyretic

- Just a special type of policy
- Forwarding to a “bucket”
  - `q = packets(limit=1,group_by=['srcip'])`
- Used like any other (>>, +)
- w/ Callback function registration
  - `q.register_callback(printer)`

Example: Monitor

```
q = count_bytes(interval=1)
q.register_callback(printer)
monitor = match(srcip='10.0.0.1') >> q
```

Query Policy Library

Syntax	Side Effect

`packets(`
`  limit=n,`
`  group_by=[f1,f2,…])`
	callback on every packet received for up to n packets identical on fields f1,f2,...

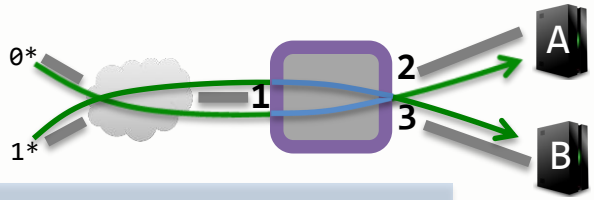
`count_packets(`
`  interval=t,`
`  group_by=[f1,f2,…])`
	count every packet received
	callback every t seconds providing count for each group

`count_bytes(`
`  interval=t,`
`  group_by=[f1,f2,…])`
	count bytes received
	callback every t seconds providing count for each group

Recall OpenFlow Toy

`Balance` `Route` `Monitor`

In Pyretic




```
balance =
  (match(srcip=0*,dstip=P) >> modify(dstip=A)) +
  (match(srcip=1*,dstip=P) >> modify(dstip=B)) +
  (~match(        dstip=P) >> identity)
```

```
route =
  (match(dstip=A) >> fwd(2)) +
  (match(dstip=B) >> fwd(3)) +
  (~(match(dstip=A) | match(dstip=B)) >> fwd(1))
```

```
b = count_bytes(interval=1)
b.register_callback(print)
monitor = match(srcip=X) >> b
```

Compared to

```
install_flowmod(5,srcip=X &  dstip=P,[mod(dstip=A), fwd(2)])
install_flowmod(4,srcip=0* & dstip=P,[mod(dstip=A), fwd(2)])
install_flowmod(4,srcip=1* & dstip=P,[mod(dstip=B), fwd(3)])
install_flowmod(4,srcip=X & dstip=A ,[              fwd(2)])
install_flowmod(4,srcip=X &  dstip=B,[              fwd(3)])
install_flowmod(3,                dstip=A,[         fwd(2)])
install_flowmod(3,                dstip=B,[         fwd(3)])
install_flowmod(2,srcip=X                 ,[        fwd(1)])
install_flowmod(1,*                       ,[        fwd(3)])
```

Programmer Wishlist

- Encode policy concisely
- Write portable code
- Specify traffic of interest
- Compose code from independent modules
- Query network state
- **Modify policy dynamically**

Functions Support Dynamism

- Dynamic policies are also functions of time
- Value at current moment in `self.policy`
- Reassigning updates dynamic policy
- Can be driven by queries

Example: MAC-Learning

New Type of Dynamic Policy

First packet with unique srcmac, switch

```
class learn(DynamicPolicy):
  def init(self):
    q = packets(limit=1,['srcmac','switch'])
    q.register_callback(update)
    self.policy = flood() + q
  def update(self,pkt):
    self.policy = if_(match(dstmac=pkt['srcmac'],
switch=pkt['switch']),
                      fwd(pkt['inport']),
                      self.policy)
```

Defined momentarily

Update current val

Initialize current value of time series

to flood

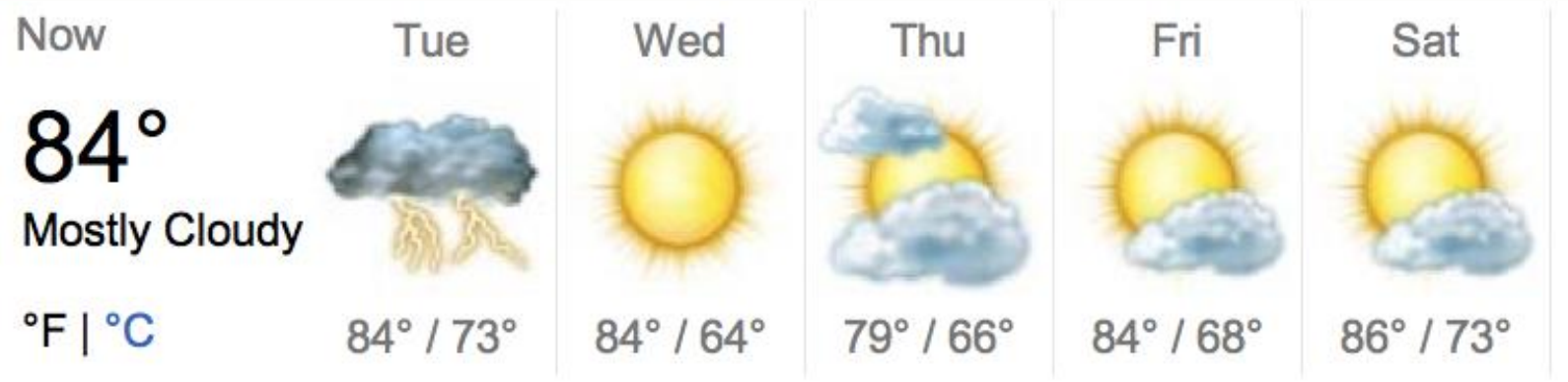
Otherwise, policy unchanged

If newly learned MAC

Forward directly to learned port

Dynamic Policies

Can also be driven by external events!



Example: UI Firewall

```
def __init__(self):
  print "initializing firewall"
  self.firewall = {}
  ...
  self.ui = threading.Thread(
              target=self.ui_loop)
  self.ui.daemon = True
  self.ui.start()
```

```
def AddRule (self, mac1, mac2):
 if (mac2,mac1) in self.firewall:
            return
 self.firewall[(mac1,mac2)]=True
 self.update_policy()
```

```
def update_policy (self):
   self.policy = ~union([(match(srcmac=mac1) &
                          match(dstmac=mac2)) |
                         (match(dstmac=mac1) &
                          match(srcmac=mac2))
                         for (mac1,mac2)
                         in self.firewall.keys()])
```

Programmer Wishlist

- Encode policy concisely
- **Write portable code**
- Specify traffic of interest
- **Compose code from independent modules**
- Query network state
- Modify policy dynamically

OpenFlow Topology-Dependence

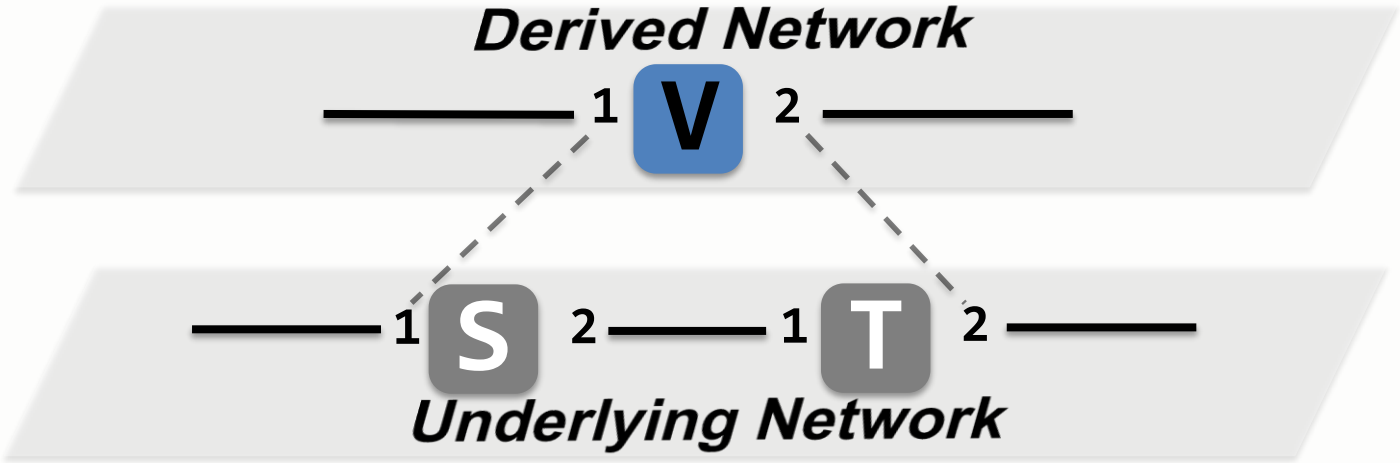
- Application written for single switch cannot easily be ported to run over a distributed collection of switches

- Or be made to share switch hardware with other packet-processing applications.

Pyretic Topology Abstraction

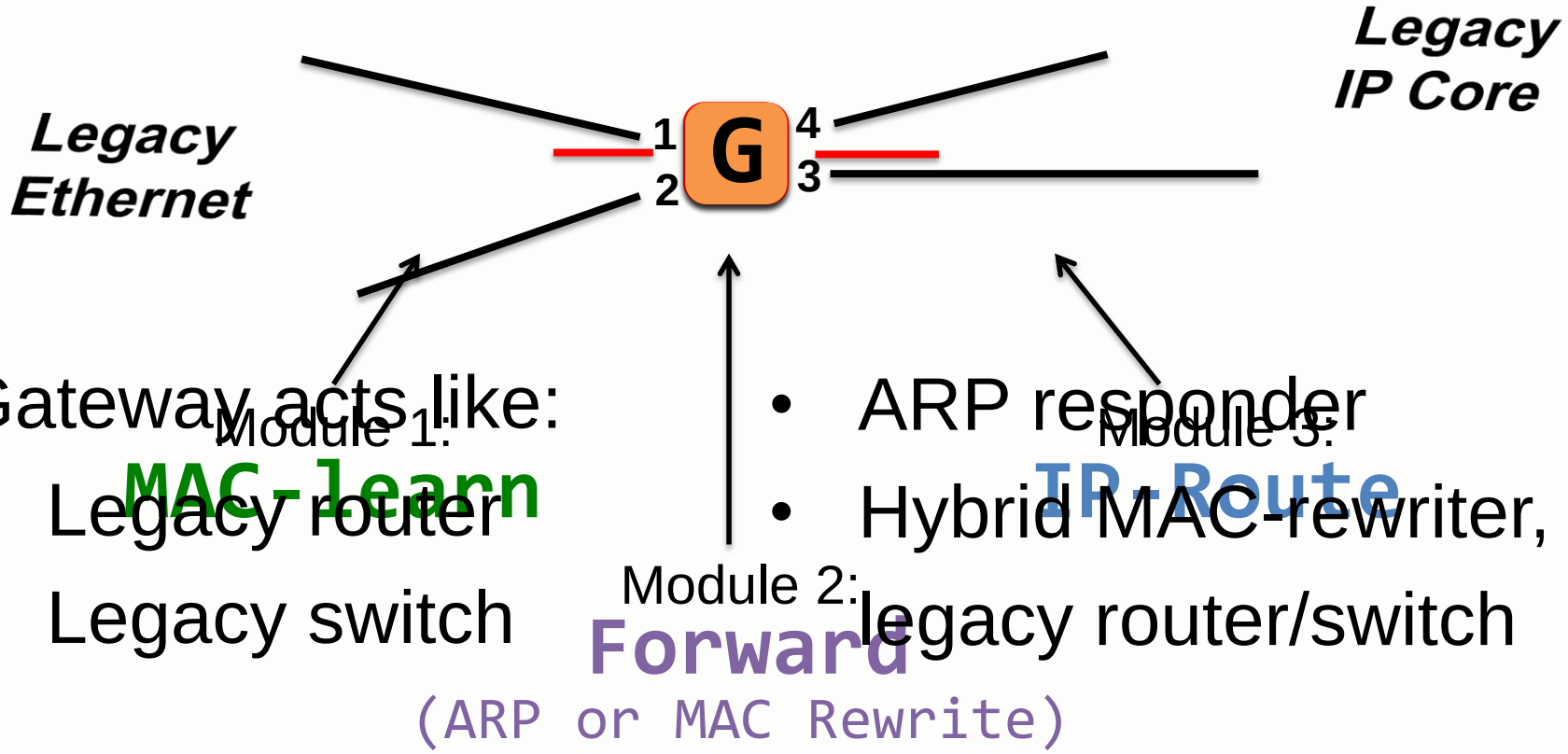
- Built as an application-level library
- On top of other features introduced
- Examples provided for
  - Merging
  - Splitting
  - Refining

Topology Abstraction: Merge



- Simplest of topology abstraction examples
- Build a distributed middlebox
  by running centralized middlebox app on V!

Topology Abstraction: Split Replace Legacy Gateway

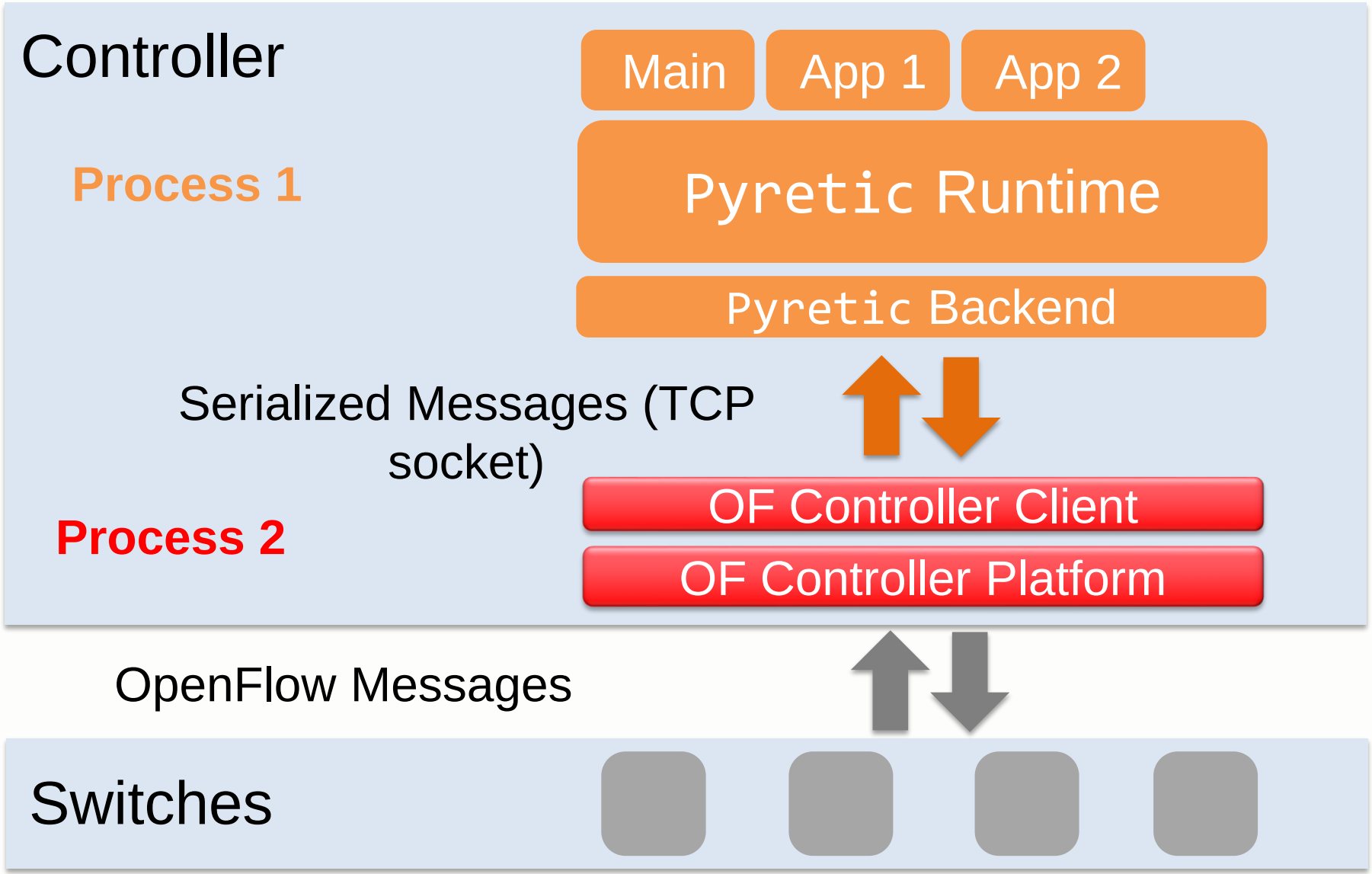


Gateway acts like:

- Legacy router
- Legacy switch
- ARP responder
- Hybrid MAC-rewriter, legacy router/switch

Pyretic Platform

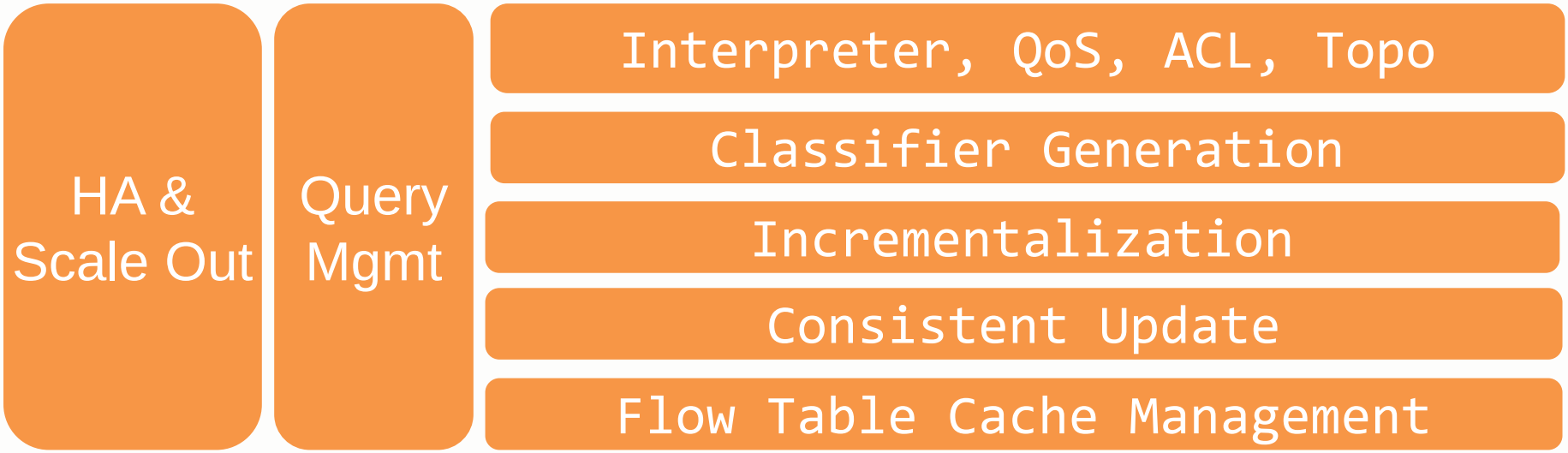
Platform Architecture



Modes of Operation

- Interpreted (on controller)
  - Fallback when rules haven't been installed
  - Good for debugging
- Reactive
  - Rules installed in response to packets seen
  - Fallback when proactive isn't feasible
  - Various flavors and optimizations
- Proactive
  - Analyze policy and push rules
  - Computation can be an issue
  - Currently working out kinks

Runtime Architecture (in Progress)



Also in Progress

- New features
  - Policy access controls
  - QoS operators
  - Enhanced querying library
- Applications
  - Incorporation of RADIUS & DHCP services
  - Wide-area traffic-management solutions for ISPs at SDN-enabled Internet Exchange Points.

Don’t Just Take Our Word

- 2013 NSDI Community Award Winner
- Featured in Coursera SDN MOOC (~50K students, over 1K did assignments)
- Georgia Tech Resonance Reimplementation
  - Approximately one programmer-day
  - Six-fold reduction in code size (prev in NOX)
  - Short expressions replaced complex code
    - Matching packets
    - Modifying and injecting packets
    - Constructing and installing OpenFlow rules
  - Added new features too complex for NOX

Test It Out Yourself

www.frenetic-lang.org/pyretic